

# 4

## EXPLOITING SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

*If you find that you’re spending all your time on theory, start turning some attention to practical things; it will improve your theories. If you find that you’re spending almost all your time on practice, start turning some attention to theoretical things; it will improve your practice.*

— Donald Knuth

**I**N this chapter, we illustrate the application of Software Diversification for both offensive and defensive purposes. We discuss two selected use cases that demonstrate practical applications of our contributions.

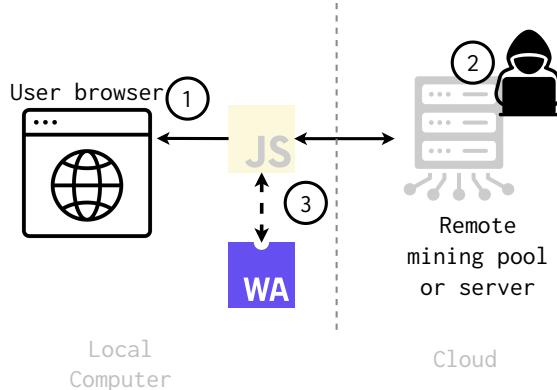
### 4.1 Offensive Diversification: Malware evasion

The primary malicious use of WebAssembly in browsers is cryptojacking [? ]. This is due to the essence of cryptojacking, the faster the mining, the better. Although Lehmann and colleagues [? ] observe a decline in browser-based cryptominers, mainly due to the shutdown of Coinhive, a 2022 report by Kaspersky indicates that the use of cryptominers is on the rise [? ].

Let us illustrate how a malicious WebAssembly binary could be involved into browser cryptojacking. Figure 4.1 illustrates a browser attack scenario: a practical WebAssembly cryptojacking attack consists of three components: a WebAssembly binary, a JavaScript wrapper, and a backend cryptominer pool. The WebAssembly binary is responsible for executing the hash calculations, which consume significant computational resources. The JavaScript wrapper facilitates the communication between the WebAssembly binary and the cryptominer pool.

---

<sup>0</sup>Compilation probe time 2023/10/25 10:48:51



**Figure 4.1:** A remote mining pool server, a JavaScript wrapper and the WebAssembly binary form the triad of a cryptojacking attack in browser clients.

The aforementioned components require the following steps to succeed in cryptomining. First, the victim visits a web page infected with the cryptojacking code. The web page establishes a channel to the cryptominer pool, which then assigns a hashing job to the infected browser. The WebAssembly cryptominer calculates thousands of hashes inside the browser. Once the malware server receives acceptable hashes, it is rewarded with cryptocurrencies for the mining. Then, the server assigns a new job, and the mining process starts over.

Both antivirus software and browsers have implemented measures to detect cryptojacking. For instance, Firefox employs deny lists to detect cryptomining activities [? ]. The academic community has also contributed to the body of work on detecting or preventing WebAssembly-based cryptojacking, as outlined in Section 2.1.6. However, malicious actors can employ evasion techniques to circumvent these detection mechanisms. Bhansali et al. are among the first who have investigated how WebAssembly cryptojacking could potentially evade detection [? ], highlighting the critical importance of this use case. For an in-depth discussion on this topic, we direct the reader to our contribution [? ]. The case illustrated in the subsequent sections uses Offensive Software Diversification for evading malware detection in WebAssembly.

#### 4.1.1 Cryptojacking defense evasion

Considering the previous scenario, several techniques can be directly implemented in browsers to thwart cryptojacking by identifying the malicious WebAssembly components. Such defense scenario is illustrated in Figure 4.2, where the WebAssembly malicious binary is blocked in ③. The primary aim of our use case is to investigate the effectiveness of code diversification as a means to



**Figure 4.2:** Cryptojacking scenario in which the malware detection mechanism is bypassed by using an evasion technique.

circumvent cryptojacking defenses. Specifically, we assess whether the following evasion workflow can successfully bypass existing security measures:

1. The user loads a webpage infected with cryptojacking malware, which leverages network resources for execution—corresponding to (1) and (2) in Figure 4.2.
2. A malware detection mechanism (malware oracle) identifies and blocks malicious WebAssembly binaries at (3). For example, a network proxy could intercept and forward these resources to an external detection service via its API.
3. Anticipating that a specific malware detection system is consistently used for defense, the attacker swiftly generates a variant of the WebAssembly cryptojacking malware designed to evade detection at (4).
4. The attacker delivers the modified binary instead of the original one (5), which initiates the cryptojacking process and compromises the browser (6). The detection method is completely oblivious to the malicious nature of the binary, and the attack is successful.

#### 4.1.2 Methodology

In this study, we categorize malware detection mechanisms as malware oracles, which can be of two types: binary and numeric. A binary oracle provides a binary decision, labeling a WebAssembly binary as either malicious or benign. In contrast, a numeric oracle returns a numerical value representing the confidence level of the detection.

**Definition 3.** *Malware oracle: A malware oracle is a detection mechanism that returns either a binary decision or a numerical value indicating the confidence level of the detection.*

For empirical validation, we employ VirusTotal as a numeric oracle and MINOS [?] as a binary oracle. VirusTotal is an online service that analyzes files and returns a confidence score in the form of the number of antivirus that flag the input file as malware, thus qualifying as a numeric oracle. MINOS, on the other hand, converts WebAssembly binaries into grayscale images and employs a convolutional neural network for classification. It returns a binary decision, making it a binary oracle.

We use the wasmbench dataset [?] to establish a ground truth. After running the wasmbench dataset through VirusTotal and MINOS, we identify 33 binaries flagged as malicious by at least one VirusTotal vendor and also detected by MINOS. Then, to simulate the evasion scenario, we use WASM-MUTATE to generate WebAssembly binary variants to evade malware detection. We use WASM-MUTATE in two configurations: controlled and uncontrolled diversification.

**Definition 4. Controlled Diversification:** *In controlled diversification, the transformation process of a WebAssembly program is guided by a numeric oracle, which influences the probability of each transformation. For instance, WASM-MUTATE can be configured to apply transformations that minimize the oracle’s confidence score. Note that controlled diversification needs a numeric oracle.*

**Definition 5. Uncontrolled Diversification:** *Unlike controlled diversification, uncontrolled diversification is a stochastic process where each transformation has an equal likelihood of being applied to the input WebAssembly binary.*

Based on the two types of malware oracles and diversification configurations, we examine three scenarios: 1) VirusTotal with a controlled diversification, 2) VirusTotal with an uncontrolled diversification, and 3) MINOS with an uncontrolled diversification. Notice that, the fourth scenario with MINOS and a controlled diversification is not feasible, as MINOS is a binary oracle and cannot provide the numerical values required for controlled diversification.

Our evaluation focuses on two key metrics: the success rate of evading detection mechanisms in VirusTotal and MINOS across the 33 flagged binaries, and the performance impact on the variants that successfully evade detection. The first metric measures the efficacy of WASM-MUTATE in bypassing malware detection systems. For each flagged binary, we input it into WASM-MUTATE, configured with the selected oracle and diversification strategy. We then iteratively apply transformations to the output from the preceding step. This iterative process is halted either when the binary is no longer flagged by the oracle or when a maximum of 1000 stacked transformations have been applied (see Definition 2). This process is repeated with 10 random seeds per binary

to simulate 10 different evasion experiments per binary. The second metric is crucial for validating the real-world applicability of WASM-MUTATE in evading malware detection. Specifically, if the evasion process significantly degrades the performance of the resulting binary compared to its original version, it becomes less likely to be employed in practical scenarios, such as cryptojacking. For this, we execute, end-to-end, the variants that fully evade VirusTotal when generated with WASM-MUTATE in controlled and uncontrolled diversification configurations. We select variants for which we could completely reproduce the three components in Figure 4.1.

### 4.1.3 Results

In Table 4.1, we present a comprehensive summary of the evasion experiments presented in [?], focusing on two oracles: VirusTotal and MINOS[?]. The table is organized into two main categories to separate the results for each malware oracle. For VirusTotal, we further subdivide the results based on the two diversification configurations we employ: uncontrolled and controlled diversification. In these subsections, the columns indicate the number of VirusTotal vendors that flag the original binary as malware (#D), the maximum number of successfully evaded detectors (Max. #evaded), and the average number of transformations required (Mean #trans.) for each sample. We highlight in bold text the values for which the uncontrolled diversification or controlled diversification setups best, the lower, the better. The MINOS section solely includes a column that specifies the number of transformations needed for complete evasion. The table has  $33 + 1$  rows, each representing a unique WebAssembly malware study subject. The final row offers the median number of transformations required for evasion across our evaluated setups and oracles.

**Uncontrolled diversification to evade VirusTotal:** We execute an uncontrolled diversification with WASM-MUTATE, setting a limit of 1000 iterations for each binary. In every iteration, we query VirusTotal to determine if the newly generated binary can elude detection. We repeat this procedure with ten distinct seeds for each binary, replicating ten different evasion experiments. As the uncontrolled diversification section of Table 4.1 illustrates, we successfully produce variants that fully evade detection for 30 out of 33 binaries. The average amount of iterations required to produce a variant that evades all detectors oscillates between 120 to 635 stacked transformations. The mean number of iterations needed never exceeds 1000 stacked transformations. However, three binaries remain detectable under the uncontrolled diversification setup. In these instances, the algorithm fails to evade 5 out of 31, 6 out of 30, and 5 out of 26 detectors. This shortfall can be attributed to the maximum number of iterations, 1000, that we employ in our experiments. Increasing iterations further, however, seems unrealistic. If certain transformations enlarge the binary size, a significantly large binary could become impractical due to

Hash	#D	VirusTotal				MINOS[? ]	
		Uncontrolled diversification		Controlled diversification			
		Max. evaded	Mean trans.	Max. evaded	Mean trans.		
47d29959	31	<b>26</b>	N/A	19	N/A	100	
9d30e7f0	30	<b>24</b>	N/A	17	N/A	419	
8ebf4e44	26	<b>21</b>	N/A	13	N/A	92	
c11d82d	20	20	<b>355</b>	20	446	115	
0d996462	19	19	<b>401</b>	19	697	24	
a32a6f4b	18	18	635	18	<b>625</b>	1	
fbdd1efa	18	18	<b>310</b>	18	726	1	
d2141ff2	9	9	<b>461</b>	9	781	81	
aafff587	6	6	484	6	<b>331</b>	1	
046dc081	6	6	404	6	<b>159</b>	33	
643116ff	6	6	<b>144</b>	6	436	47	
15b86a25	4	4	253	4	<b>131</b>	1	
006b2fb6	4	4	<b>282</b>	4	380	1	
942be4f7	4	4	200	4	200	29	
7c36f462	4	4	236	4	<b>221</b>	85	
fb15929f	4	4	<b>297</b>	4	475	1	
24aae13a	4	4	<b>252</b>	4	401	980	
000415b2	3	3	302	3	<b>34</b>	960	
4cdbbbb1	3	3	295	3	<b>72</b>	1	
65debcbe	2	2	131	2	<b>33</b>	38	
59955b4c	2	2	130	2	<b>33</b>	38	
89a3645c	2	2	431	2	<b>107</b>	108	
a74a7cb8	2	2	124	2	<b>33</b>	38	
119c53eb	2	2	104	2	<b>18</b>	1	
089dd312	2	2	153	2	<b>123</b>	68	
c1be4071	2	2	130	2	<b>33</b>	38	
dceaf65b	2	2	140	2	132	66	
6b8c7899	2	2	143	2	<b>33</b>	38	
a27b45ef	2	2	145	2	<b>33</b>	33	
68ca7c0e	2	2	137	2	<b>33</b>	38	
f0b24409	2	2	127	2	<b>11</b>	33	
5bc53343	2	2	118	2	<b>33</b>	33	
e09c32c5	1	1	<b>120</b>	1	488	15	
Median			218		131	38	

**Table 4.1:** The table has two main categories for each malware oracle, corresponding to the two oracles we use: VirusTotal and MINOS. For VirusTotal, divide the results based on the two diversification configurations: uncontrolled and controlled diversification. We provide columns that indicate the number of VirusTotal vendors that flag the original binary as malware (#D), the maximum number of successfully evaded detectors (Max. #evaded), and the average number of transformations required (Mean #trans.) for each sample. We highlight in bold text the values for which diversification setups are best, the lower, the better. The MINOS section includes a column that specifies the number of transformations needed for complete evasion. The final row offers the median number of transformations required for evasion across our evaluated setups and oracles.

bandwidth limitations. In summary, uncontrolled diversification with WASM-MUTATE markedly reduces the detection rate by VirusTotal antivirus vendors for cryptojacking malware, achieving total evasion in 30 out of 33 (90%) cases within the malware dataset. WASM-MUTATE proves capable of successfully evading detection systems in just a few minutes.

**Controlled diversification to evade VirusTotal:** Uncontrolled diversification does not guide the diversification based on the number of evaded detectors, it is purely random, and has some drawbacks. For example, some transformations might suppress other transformations previously applied. We have observed that, by carefully selecting the order and type of transformations applied, it is possible to evade detection systems in fewer iterations. This can be appreciated in the results of the controlled diversification part of Table 4.1. Analyzing the data in Table 4.1, we observe that the controlled diversification setup successfully generates variants that totally evade the detection for 30 out of 33 binaries, it is thus as good as the uncontrolled setup. The iterations needed for the controlled diversification setup are 92% of the iterations needed on average for the uncontrolled diversification setup. For 21 of 30 binaries that evade detection entirely, we observe that the mean number of oracle calls needed is lower than those in the baseline evasion algorithm. For example, f0b24409 needs 11 oracle calls with controlled diversification setup to fully evade VirusTotal, while for the uncontrolled one, it needs 127 oracles calls. For those 21 binaries, it needs only 40% of the calls the controlled diversification setup needs. The controlled diversification setup, having fewer stacked transformations overall, demonstrates larger efficiency than the uncontrolled diversification setup. For instance, a potential attacker may be limited by budget on the number of transformations applicable to the malware binary, e.g., the number of queries to VirusTotal. Additionally, the performance of the resulting binary benefits from this approach.

**Uncontrolled diversification to evade MINOS:** Relying exclusively on VirusTotal for detection could pose issues, particularly given the existence of specialized solutions for WebAssembly, which differ from the general-purpose vendors within VirusTotal. In Section 2.1.6 we highlight several examples of such solutions. Yet, for its simplicity, we extend this experiment by using MINOS[?], an antivirus specifically designed for WebAssembly. We incrementally apply stacked transformations to the original malware binary until MINOS is completely evaded or the maximum iteration limit is achieved. We repeat this procedure ten times for every binary. The results of these trials can be seen in the final column of Table 4.1. The bottom row of Table 4.1 highlights that fewer iterations are required to evade MINOS than VirusTotal through WebAssembly diversification, indicating a greater ease in eluding MINOS. The median iteration count needed to evade VirusTotal is 218 for the uncontrolled diversification setup, and 131 for the controlled diversification setup; however, only 38 iterations are needed for MINOS. Interestingly, WASM-MUTATE managed to evade detection for 8

out of 33 binaries in just a single iteration when testing against MINOS. This suggests the MINOS model’s vulnerability to binary diversification. We have also noted an interesting phenomenon. Based on our data, VirusTotal appears to outperform MINOS in terms of detecting cryptojacking. The primary reason for this is VirusTotal’s use of a wider array of antivirus vendors, encompassing several diverse detection approaches. In contrast, MINOS operates as a standalone detector. Consequently, this supports the employment of multiple malware oracles (meta-oracles) for the detection of cryptojacking malware in browsers, even when it is possible a Wasm-specific detection mechanism.

**WebAssembly execution impact:** To evaluate the real-world efficacy of WASM-MUTATE in evading malware detection, we focused on six binaries that we could build and execute end-to-end, as these had all three components outlined in Figure 4.1. We select only six binaries because the process of building and executing the binaries involves three components. As detailed in Figure 4.1, they include the binary, its JavaScript complement, and the miner pool. These components were not found for the remaining 27 binaries in the study subjects. For the six binaries, we then replace the original WebAssembly code with variants generated using VirusTotal as the malware oracle and WASM-MUTATE for both controlled and uncontrolled diversification configurations. We then execute both the original and the generated variants. We assessed the variants based on the hash generation rate, providing a hint on the impact of diversification in the WebAssembly variants execution.

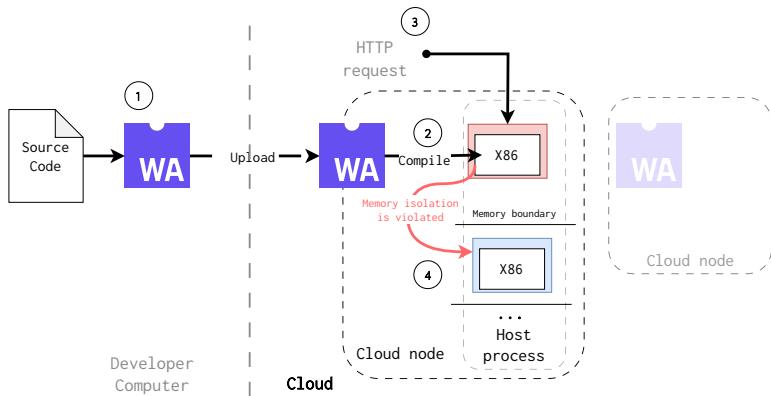
We have found that 19% of the generated variants outperformed the original cryptojacking binaries. This improvement is attributed to WASM-MUTATE’s ability to introduce code optimizations. Additionally, debloating transformations, which eliminate unnecessary structures and dead code, resulted in a higher hash generation rate during the initial seconds of mining, likely due to faster compilation times. This suggests that focused optimization serves as a valuable tool for evasion in browsers. On the contrary, 80% of the generated variants are less efficient than the original binary, with the least efficient variant operating at only 20% of the original hash generation rate. This performance drop is primarily due to non-optimal transformations introduced by WASM-MUTATE. Variants generated through uncontrolled diversification are generally slower. In summary, controlled diversification yielded variants that evaded VirusTotal detection with minimal performance overhead—the worst-performing variant was only 1.93 times slower than the original.

Contribution paper

Our work provides evidence that the malware detection community has opportunities to strengthen the automatic detection of cryptojacking WebAssembly malware. The results of our work are actionable, as we also provide quantitative evidence on specific malware transformations on which detection methods can focus. The case discussed in this section is fully detailed in Cabrera-Arteaga et al. "WebAssembly Diversification for Malware Evasion" at *Computers & Security*, 2023 <https://www.sciencedirect.com/science/article/pii/S0167404823002067>.

## 4.2 Defensive Diversification: Speculative Side-channel protection

As discussed in Section 2.1, WebAssembly is quickly becoming a cornerstone technology in backend systems. Leading companies like Cloudflare and Fastly are championing the integration of WebAssembly into their edge computing platforms, thereby enabling developers to deploy applications that are both modular and securely sandboxed. These server-side WebAssembly applications are generally architected as isolated, single-responsibility services, a model referred to as Function-as-a-Service (FaaS) [? ? ]. The operational flow of WebAssembly binaries in FaaS platforms is illustrated in Figure 4.3.



**Figure 4.3:** WebAssembly binaries on FaaS platforms. Developers can submit any WebAssembly binary to the platform to be executed as a service in a sandboxed and isolated manner. Yet, WebAssembly binaries are not immune to Spectre attacks.

The fundamental advantage of using WebAssembly in FaaS platforms lies in its ability to encapsulate thousands of client WebAssembly binaries within

a singular host process. A developer could compile its source code into a WebAssembly program suitable for the cloud platform and then submit it (① in Figure 4.3). This host process is then disseminated across a network of servers and data centers (② in Figure 4.3). These platforms convert WebAssembly programs into native code, which is subsequently executed in a sandboxed environment. Host processes can then instantiate new WebAssembly sandboxes for each client function, executing them in response to specific user requests with nanosecond-level latency (③ in Figure 4.3). This architecture inherently isolates WebAssembly binary executions from each other as well as from the host process, enhancing security.

However, while WebAssembly is engineered with a strong focus on security and isolation, it is not entirely immune to vulnerabilities such as Spectre attacks [? ?] (④ in Figure 4.3). In the sections that follow, we explore how software diversification techniques can be employed to harden WebAssembly binaries against such attacks. For an in-depth discussion on this topic, we direct the reader to our contribution [?].

#### 4.2.1 Threat model: speculative side-channel attacks

To illustrate the threat model concerning WebAssembly programs in FaaS platforms, consider the following scenarios. Developers, including potentially malicious actors, have the ability to submit any WebAssembly binary to the FaaS platform. A malicious actor could then upload a WebAssembly binary that, once compiled to native code, employs Spectre attacks to either leak sensitive information from the host process or violate Control Flow Integrity (CFI). Furthermore, even if a submitted WebAssembly binary is not intentionally malicious, it may still be vulnerable to Spectre attacks. For instance, a malicious actor could exploit this vulnerability by executing the susceptible binary through the FaaS service.

Spectre attacks exploit hardware-based prediction mechanisms to trigger mispredictions, leading to the speculative execution of specific instruction sequences that are not part of the original, sequential execution flow. By taking advantage of this speculative execution, an attacker can potentially access sensitive information stored in the memory allocated to other WebAssembly instances (including itself) or even the host process itself. This poses a significant risk, compromising both the security and integrity of the overall system.

Narayan and colleagues [?] have categorized potential Spectre attacks on WebAssembly binaries into three distinct types, each corresponding to a specific hardware predictor being exploited and a particular FaaS scenario: Branch Target Buffer Attacks, Return Stack Buffer Attacks, and Pattern History Table Attacks defined as follows:

1. The Spectre Branch Target Buffer (btb) attack exploits the branch target buffer by predicting the target of an indirect jump, thereby rerouting

Program	Attack
btb_breakout	Spectre branch target buffer (btb)
btb_leakage	Spectre branch target buffer (btb)
ret2spec	Spectre Return Stack Buffer (rsb)
pht	Spectre Pattern History Table (pht)

**Table 4.2:** WebAssembly program name and its respective attack.

speculative control flow to an arbitrary target.

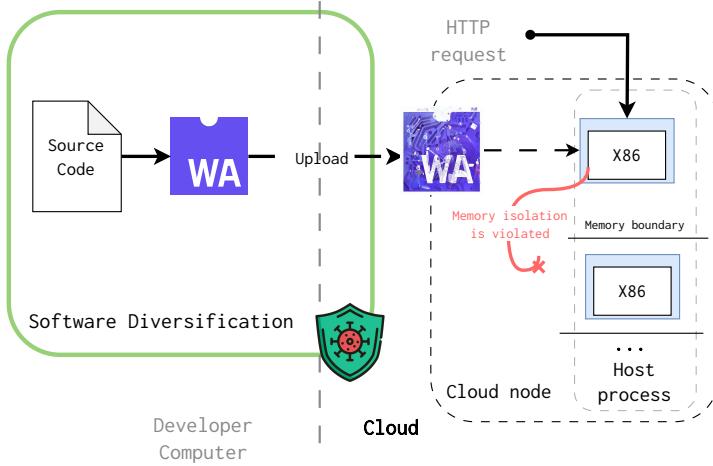
2. The Spectre Return Stack Buffer (rsb) attack exploits the return stack buffer that stores the locations of recently executed call instructions to predict the target of `ret` instructions.
3. The Spectre Pattern History Table (pht) takes advantage of the pattern history table to anticipate the direction of a conditional branch during the ongoing evaluation of a condition.

#### 4.2.2 Methodology

Our goal is to empirically validate that Software Diversification can effectively mitigate the risks associated with Spectre attacks in WebAssembly binaries. The green-highlighted section in Figure 4.4 illustrates how Software Diversification can be integrated into the FaaS platform workflow. The core idea is to generate unique and diverse WebAssembly variants that can be randomized at the time of deployment. For this use case, we employ WASM-MUTATE as our tool for Software Diversification.

To empirically demonstrate that Software Diversification can indeed mitigate Spectre vulnerabilities, we reuse the WebAssembly attack scenarios proposed by Narayan and colleagues in their work on Swivel [?]. Swivel is a compiler-based strategy designed to counteract Spectre attacks on WebAssembly binaries by linearizing their control flow during machine code compilation. Our approach differs from theirs in that it is binary-based, compiler-agnostic, and platform-agnostic; we do not propose altering the deployment or toolchain of FaaS platforms.

To measure the efficacy of WASM-MUTATE in mitigating Spectre, we diversify four WebAssembly binaries proposed in the Swivel study. The names of these programs and the specific attacks we examine are available in Table 4.2. For each of these four binaries, we generate up to 1000 random stacked transformations (see Definition 2) using 100 distinct seeds, resulting in a total of 100,000 variants for each original binary. At every 100th stacked transformation for each binary and seed, we assess the impact of diversification on the Spectre attacks by measuring the attack bandwidth for data exfiltration. This metric not



**Figure 4.4:** Diversifying WebAssembly binaries to mitigate Spectre attacks in FaaS platforms.

only captures the success or failure of the attacks but also quantifies the extent to which data exfiltration is hindered. For example, a variant that still leaks data but does so at an impractically slow rate would be considered hardened against the attack.

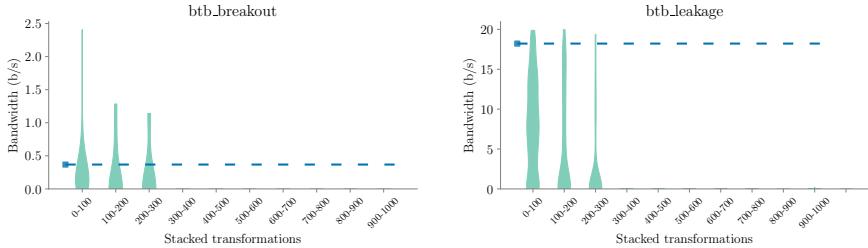
**Definition 6.** *Attack bandwidth:* Given data  $D = \{b_0, b_1, \dots, b_C\}$  being exfiltrated in time  $T$  and  $K = k_1, k_2, \dots, k_N$  the collection of correct data bytes, the bandwidth metric is defined as:

$$\frac{|b_i \text{ such that } b_i \in K|}{T}$$

### 4.2.3 Results

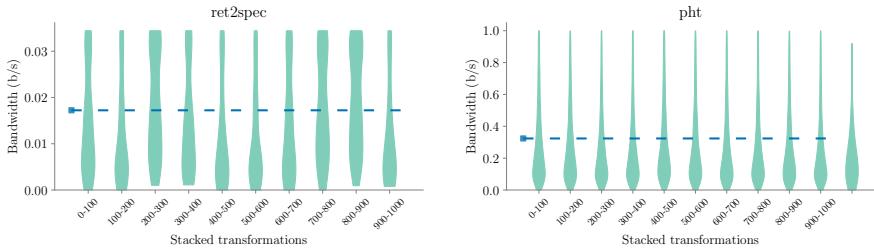
Figure 4.5 offers a graphical representation of WASM-MUTATE’s influence on the Swivel original programs: `btb_breakout` and `btb_leakage` with the `btb` attack. The Y-axis represents the exfiltration bandwidth (see Definition 6). The bandwidth of the original binary under attack is marked as a blue dashed horizontal line. In each plot, the variants are grouped in clusters of 100 stacked transformations. These are indicated by the green violinplots.

**Population Strength:** For the binaries `btb_breakout` and `btb_leakage`, WASM-MUTATE exhibits a high level of effectiveness, generating variants that leak less information than the original in 78% and 70% of instances, respectively. For both programs, after applying 200 stacked transformations, the exfiltration bandwidth drops to zero. This implies that WASM-MUTATE is capable of



**Figure 4.5:** Impact of WASM-MUTATE over *btb\_breakout* and *btb\_leakage* binaries. The Y-axis denotes exfiltration bandwidth, with the original binary’s bandwidth under attack highlighted by a blue marker and dashed line. Variants are clustered in groups of 100 stacked transformations, denoted by green violinplots. Overall, for all 100000 variants generated out of each original program, 70% have less data leakage bandwidth. After 200 stacked transformations, the exfiltration bandwidth drops to zero.

synthesizing variants that are entirely protected from the original attack. If we consider the results in Table 3.1, generating a variant with 200 stacked transformations can be accomplished in just a matter of seconds for a single WebAssembly binary. When scaled to the scope of a global FaaS platform, this means that a unique, hardened variant could be deployed for each machine and even for each fresh WebAssembly spawned per user request.



**Figure 4.6:** Impact of WASM-MUTATE over *ret2spec* and *pht* binaries. The Y-axis denotes exfiltration bandwidth, with the original binary’s bandwidth under attack highlighted by a blue marker and dashed line. Variants are clustered in groups of 100 stacked transformations, denoted by green violinplots. Overall, for both programs approximately 70% of the variants have less data leakage bandwidth.

As illustrated in Figure 4.6, similarly to Figure 4.5, WASM-MUTATE significantly impacts the programs *ret2spec* and *pht* when subjected to their respective attacks. In 76% of instances for *ret2spec* and 71% for *pht*, the generated variants demonstrated reduced attack bandwidth compared to the

original binaries. The plots reveal that a notable decrease in exfiltration bandwidth occurs after applying at least 100 stacked transformations. While both programs show signs of hardening through reduced attack bandwidth, this effect is not immediate and requires a substantial number of transformations to become effective. Additionally, the bandwidth distribution is more varied for these two programs compared to the two previous ones. Our analysis suggests a correlation between the reduction in attack bandwidth and the complexity of the binary being diversified. Specifically, `ret2spec` and `pht` are substantially larger programs, containing over 300,000 instructions, compared to `btb_breakout` and `btb_leakage`, which have fewer than 800 instructions. Therefore, given that WASM-MUTATE performs one transformation per invocation, the probability of affecting critical components to hinder attacks decreases in larger binaries.

**Managed memory impact:** The success in diminishing Spectre attacks is explained by the fact that WASM-MUTATE synthesizes variants that effectively alter memory access patterns. We have identified four primary factors responsible for the divergence in memory accesses among WASM-MUTATE generated variants. First, modifications to the binary layout—even those that do not affect executed code—inevitably alter memory accesses within the program’s stack. Specifically, WASM-MUTATE generates variants that modify the return addresses of functions, which consequently leads to differences in execution flow and memory accesses. Second, one of our rewriting rules incorporates artificial global values into WebAssembly binaries. The access to these global variables inevitably affects the managed memory (see Section 2.1.4). Third, WASM-MUTATE injects ‘phantom’ instructions which do not aim to modify the outcome of a transformed function during execution. These intermediate calculations trigger the spill/reload component of the wasmtime compiler, varying spill and reload operations. In the context of limited physical resources, these operations temporarily store values in memory for later retrieval and use, thus creating diverse managed memory accesses (see the example at Section 3.3.1). Finally, certain rewriting rules implemented by WASM-MUTATE replicate fragments of code, e.g., performing commutative operations. These code segments may contain memory accesses, and while neither the memory addresses nor their values change, the frequency of these operations does.

**Contribution paper**

WASM-MUTATE crafts WebAssembly binaries that are resilient to Spectre-like attacks. By integrating a software diversification layer into WebAssembly binaries deployed on Function-as-a-Service (FaaS) platforms, security can be significantly bolstered. This approach allows for the deployment of unique and diversified WebAssembly binaries, potentially utilizing a distinct variant for each cloud node, thereby enhancing the overall security. The case discussed in this section is fully detailed in Cabrera-Arteaga et al. "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly" *Under review* <https://arxiv.org/pdf/2309.07638.pdf>.

### 4.3 Intrinsic properties of diversification

In Section 4.2, we have noted an increasing trend of exfiltration bandwidth in certain variants. Section 4.1 presents a similar case, indicating that without a clear objective in the diversification process, uncontrolled diversification can be counterproductive. This implies that not all transformations contribute equally to the diversification objectives of WebAssembly. In the following, we elaborate on three cornerstones properties for improving Software Diversification with relation to WebAssembly.

**Preservation:** Some transformations yield distinct WebAssembly binaries, yet their JIT compilation produces identical machine code. Non-preserved transformations undermine the effectiveness of diversification, as discussed in Section 3.4. Incorporating random `nop` operations directly into WebAssembly, for instance, does not alter the final machine code because JIT compilers typically eliminate these `nop` operations. This phenomenon is also observed with transformations to the custom sections of WebAssembly binaries. Identical machine code, even when their WebAssembly variants are different, can be detected by malware detectors. For practitioners, malware detection tools can be enhanced by incorporating a pre-compilation step to normalize WebAssembly binaries. Besides, developers could focus on transformations that preserve the machine code, as they are more likely to contribute to the diversification objectives, e.g., evasion. On the other hand, side-channel attacks occur at the machine code level. Thus, making WebAssembly variants preservation is essential code for successful defensive diversification, i.e., if the machine code is preserved, the side-channel attack will not be effective against the WebAssembly variant.

**Disrupting timers:** Cache timing side-channel attacks, including for the four binaries analyzed in Section 4.2, depend on precise timers to measure cache access times. Disrupting these timers can effectively neutralize the attack [?]

]. The WebAssembly variants inherently adopt a similar approach, introducing perturbations in the timing steps of WebAssembly variants. This is illustrated in Listing 4.1 and Listing 4.2, where the former shows the original time measurement and the latter presents a variant with introduced operations. By introducing additional instructions, the inherent randomness in the time measurement of a single or a few instructions is amplified, thereby reducing the timer’s accuracy.

```
;; Code from original btb_breakout
...
(call $readTimer)
(set_local $end_time)
... access to mem
(i64.sub (get_local $end_time) (get_local $start_time))
(set_local $duration)
...
```

---

*Listing 4.1:* Wasm timer code.

```
;; Variant code
...
(call $readTimer)
(set_local $end_time)
<inserted instructions>
... access to mem
<inserted instructions>
(i64.sub (get_local $end_time) (get_local $start_time))
(set_local $duration)
...
```

---

*Listing 4.2:* WebAssembly variant with more instructions added in between time measurement.

**TODO** Recheck term-citation padding here

**Padding speculated instructions:** CPUs have a limit on the number of instructions they can cache. Diversification injects instructions to potentially exceed this limit, effectively disabling the speculative execution of memory accesses. This approach is akin to padding [? ], as demonstrated in Listing 4.3 and Listing 4.4. This padding disrupts the binary code’s layout in memory, hindering the attacker’s ability to initiate speculative execution. Even if speculative execution occurs, the memory access does not proceed as the attacker intended.

---

```
;; Code from original btb_breakout
...
;; train the code to jump here (index 1)
(i32.load (i32.const 2000))
(i32.store (i32.const 83)) ; just prevent optimization
...
;; transiently jump here
(i32.load (i32.const 339968)) ; S(83) is the secret
(i32.store (i32.const 83)) ; just prevent optimization
```

---

*Listing 4.3:* Two jump locations. The top one trains the branch predictor, the bottom one is the expected jump that exfiltrates the memory access.

```
;; Variant code
...
;; train the code to jump here (index 1)
<inserted instructions>
(i32.load (i32.const 2000))
<inserted instructions>
(i32.store (i32.const 83)) ; just prevent optimization
...
;; transiently jump here
<inserted instructions>
(i32.load (i32.const 339968)) ; "S"(83) is the secret
<inserted instructions>
(i32.store (i32.const 83)) ; just prevent optimization
...
```

---

*Listing 4.4:* WebAssembly variant with more instructions added indindinctly between jump places.

## ■ Conclusions

In this chapter, we discuss two sides of Software Diversification as applied to WebAssembly: Offensive Software Diversification and Defensive Software Diversification. Offensive Software Diversification underscores both the capabilities and the latent security risks inherent in applying Software Diversification to WebAssembly malware. Our research indicates that there are ways for enhancing the automated detection of cryptojacking malware in WebAssembly, e.g. by stressing their resilience with WebAssembly malware variants. On the other hand, Defensive Software Diversification acts as a preemptive safeguard, specifically to mitigate the risks posed by Spectre attacks. In the subsequent chapter, we will consolidate the principal conclusions of this dissertation and describe directions for future research.

